THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

What Stories Should We Tell Novice Prolog Programmers?

Citation for published version:
Bundy, A & Helen, P 1987, 'What Stories Should We Tell Novice Prolog Programmers?'. in R Hawley (ed.), Artificial Intelligence Programming Environments. Ellis Horwood.

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
Artificial Intelligence Programming Environments

General rights
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 15. Dec. 2015

WHAT STORIES SHOULD WE TELL NOVICE PROLOG PROGRAMMERS?

Helen Pain & Alan Bundy

D.A.I. RESEARCH PAPER NO. 269

To appear in the *"Artificial Intelligence Programming Environments Book"*
Ed. R Lawley
Published by Wiley, 1985.

Copyright (c) Helen Pain & Alan Bundy, 1985.

What Stories Should We Tell Novice Prolog Programmers?

by
Helen Pain and Alan Bundy

Acknowledgements
This research was supported by E.S.R.C. grant number H00230012.

We would also like to thank Lincoln Wallen, Dave Plummer and Paul Brna for their invaluable comments and discussion of this paper.

1. Introduction

When designing programming environments, we need to consider the best ways to explain to the user how their programs behave.

To understand the execution of a program, a model of the programming language is required. For Prolog it is necessary to explain such features as backtracking, recursion and unification. In order to do so, the model should include information about the search space, the flow of control through the search space and the Prolog clauses. This model is equivalent to du Boulay and O'Shea's 'virtual machine' in Logo [du Boulay and O'Shea 78], and to Lawrence Byrd's 'notional machine' [Byrd 80]. There are a number of different ways of representing the model. The way we choose will depend upon the context and the audience. Each of the "different ways of representing the model" will be referred to as a **Prolog story**.

An ideal Prolog story would have the following properties:

1. the flow of control through the search space would be indicated;
2. the overall search space of the call would be conveyed, in particular, the backtracking points would be indicated, and it would be obvious when ultimate success had been attained;
3. each subgoal would be displayed;
4. the unifiers produced by the resolution of subgoals would be displayed;
5. the remaining subgoals would be displayed;
6. the final instantiation of the original goal would be displayed;
7. different instantiations of a clause would be distinguished;
8. the effect of a cut, on the search space, would be indicated;
9. the clauses that resolve the subgoal away would be displayed;
10. the other clauses that could resolve with the selected subgoal would be displayed.

These properties can be seen to represent different aspects of the model. The first represents the information for searching the tree. The final two represent the database of clauses. The rest represent the search space.

When teaching Prolog, the choice of story is particularly important. The students need a good understanding of what the computer will do with their programs. They must be able to anticipate the effect of running the programs, in order to write, debug and modify them. The students might be left to infer the model for themselves, or it might be taught explicitly. The latter reduces the likelihood of inferring an incorrect or inadequate model.

The story that is told about this model should be simple to understand and use, even by people with no previous computing/mathematical experience. All stories used by Prolog teachers, primers, trace messages, error messages, etc. must be consistent with this model, though each might only represent part of the model.

This chapter is based on our experience of teaching Prolog to novice programmers. However, the issues discussed are equally relevant to all Prolog users.

Issues relating to whether the declarative or the procedural semantics should be used to describe the behaviour of a Prolog program will not be discussed in detail here. Depending on which is used, the language (or terminology) used to tell the story will differ. Keeping consistency in the language used is desirable but not easy. It is suggested that both the procedural and declarative interpretations of stories are needed; the procedural and declarative in order to write programs in Prolog and the declarative to check the logic of the programs written.

The terminology used to tell Prolog stories has been taken both from that of problem solving and from that of theorem proving. It is not feasible to use one set of terminology to the exclusion of the other, although for certain concepts terms from one or the other may be more suitable. For example, 'subgoal' is a term from problem solving which has no directly corresponding term in theorem proving (though it has been adopted by theorem proving); 'clause' from theorem proving corresponds to both rules and facts in problem solving terms; 'unifying' derives from theorem proving, but is not equivalent to the theorem proving definition. In teaching, thought needs to be given to the 'Prolog terminology' used. Terminology from both theorem proving and problem solving might be mixed, but only one term should be used for each concept to avoid causing confusion.

2. Modes of Presentation

The mode of presentation affects the ease with which a story can be told. Some stories lend themselves to presentation at the blackboard: others require extensive computing support. The three main modes of presentation that will be considered here are:

(a) blackboard mode: presentation of Prolog programs in the classroom; representation using the blackboard, overhead slide projector, talk and chalk, etc.;

(b) computer mode: presentation at a computer terminal; running, tracing, debugging programs;

(c) text mode: presentation through static written materials, such as text books.

Clearly, there is some overlap between these modes.* Whilst it may be convenient to tell different stories in different modes of presentation, we must ensure that these stories are consistent with each other. A number of different stories are currently used in Prolog books and trace messages. The first edition of Programming in Prolog [Clocksin and Mellish 81] is a particularly rich source.

3. An Example Program

Four Prolog stories will be used to give an account of the running of an example program. The strengths and weaknesses of these stories will be considered. We are assuming Prolog programs executed with left-right, depth-first control strategy, using Edinburgh syntax. The example program involves unification, backtracking and recursion. The program is given in figure 3-1. (Lines are labelled for ease of reference.)

```
l1.     location(Person, Place):-
            at(Person, Place).
l2.     location(Person, Place):-
            visit(Person, Other),
            location(Other, Place).

a1.     at(alan, room19).
a2.     at(jane, room54).
a3.     at(betty, office).

v1.     visit(dave, alan).
v2.     visit(janet, betty).
v3.     visit(lincoln, dave)

   location(lincoln,Where).
```

Figure 3-1 An Example Program and Goal: 'location'

Whilst this simple example does not involve the use of either the cut, or of building up structures to pass back a result, or of evaluable predicates, it is complex enough to illustrate the points to be made here.

The interpretation of the goal 'location(Person,Place)' is that a Person is located at a Place. The program consists of two non-unit clauses with predicate 'location' and a number of unit clauses, or assertions, with predicates 'at' and 'visit'. The location procedure can be read declaratively as:

* a person is located at a place if the person is at the place.

* a person is located at a place if the person is visiting another person and that person is located at the place.

It can also be read procedurally as:

*Video presentation might be considered to be another mode: it will not be considered separately

* to find where a person is located find what place they are at.

* to find where a person is located find who they are visiting and then find where the person they are visiting is located.

Depending upon the goal there may be more than one solution; that is, the proof may be obtained by more than one means.

4. Prolog Stories

Four stories or representations that will be considered are:

1. AND/OR trees
2. OR trees
3. Arrow diagrams
4. Byrd boxes

Other stories include the 'Flow of Satisfaction', the 'Full Trace' and the 'Partial Tree' representation [Clocksin and Mellish 81].

4.1. AND/OR Trees

A standard story, which is inherited from resolution theorem proving, is based on the AND/OR tree. This is described in more detail by Kowalski [Kowalski 79].

The AND/OR tree is the search tree of subgoals required to satisfy some initial goal. The root node is labelled by the initial goal: each ancestor node represents a single subgoal. Conjunctions and disjunctions are represented explicitly.

According to this story, Prolog searches this tree in a depth-first manner, choosing the left-most, unexplored branch. For OR choices this left-most choice reflects the top-down ordering of clauses. For AND choices it reflects the left/right ordering of subgoals. Each node is labelled with the subgoal matched at that stage, and each link is labelled with the number of the matching clause and the most general unifier.

The tree for the above example is given in figure 4- Conjoined goals are marked with a small arc in the figure.

The AND/OR tree story of the example program presents different problems according to the mode of presentation. For this particular example with the goal 'location(lincoln,Where)' the tree is not very large. It can easily be represented in blackboard mode, on an overhead slide or on the blackboard. It is also straightforward to present it in written text. On a standard v.d.u. it is more difficult: this example will just fit on the screen if it is "squashed" a little. In general it is more difficult to present graphical representations on a standard v.d.u., though using a bit-mapped display would help solve this. A better solution would be not to try to put the complete tree on the screen, or on the blackboard, but to have access to parts of it. Debugging and tracing packages currently exist that permit access to information in the AND/OR tree without displaying the complete tree.

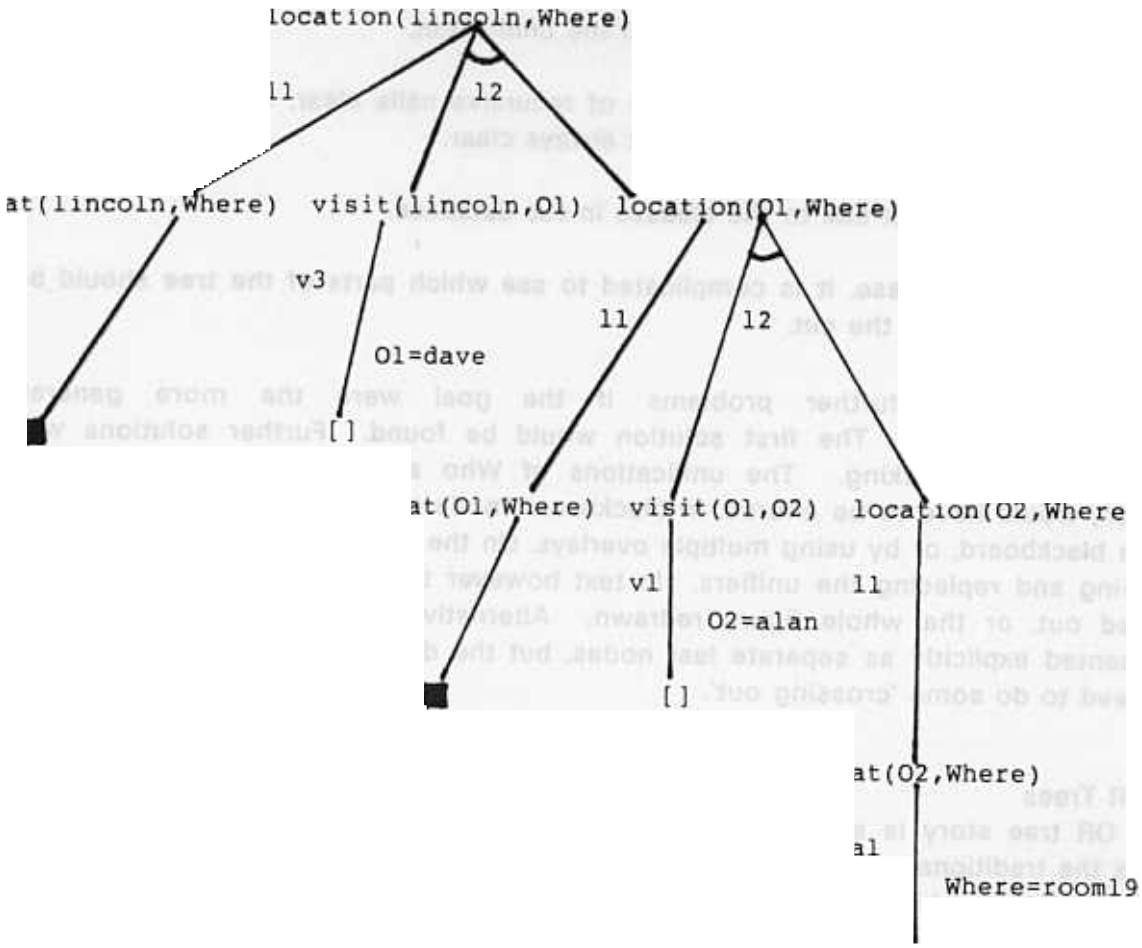

Figure 4-1 An AND/OR Tree

The AND/OR tree shows each subgoal only once. The overall search space is displayed; backtracking points are clear; the number of the clause that resolves with each subgoal is marked and the unification is displayed. The effect of the cut is not displayed in this example, but might be indicated by scribbling out the outstanding OR choices and the trees they dominate when the ! subgoal is satisfied.

The disadvantages of using the AND/OR tree are:

* It is not immediately clear when a call has been successful: this call will be represented by a subtree of the complete search tree, rather than just a branch. Highlighting this subtree by some means could make this clearer.

* It is difficult to see what subgoals are outstanding at any moment: the 'current goal' is not immediately obvious, though imposing some form of marking on the tree might help this.

* The output substitution is not clearly displayed, but must be calculated by combining the unifiers along the winning branches. If 'building a structure to return a result' is involved there would be the same problem of passing back

the value and labelling the arcs with the unification.

* To keep the different environments of recursive calls clear, the variables have to be renamed: their origins are not always clear.

* There is no direct link to the clauses in the database.

* In the general case, it is complicated to see which parts of the tree should be scribbled out by the cut.

There would be further problems if the goal were the more general goal 'location(Who,Where)'. The first solution would be found. Further solutions would be obtained by backtracking. The unifications of Who and Where, labelled for the first solution, would have to be altered. In blackboard mode this can be done: by 'rubbing out' on the blackboard, or by using multiple overlays. On the computer it can be represented by removing and replacing the unifiers. In text however the previous instantiations must be crossed out, or the whole figure redrawn. Alternatively, each match for 'at' could be represented explicitly as separate leaf nodes, but the diagram gets very cluttered and we still need to do some 'crossing out'.

4.2. OR Trees

The OR tree story is also a standard story, inherited from resolution theorem proving. This is the traditional way of describing linear resolution search spaces. Prolog is based on LUSH resolution, which is a linear resolution inference system. OR trees are described in [Kowalski 79].

The OR tree may be seen as a collapsed AND/OR tree. For each subgoal any conjoined goals are carried around in the tree, as a stack of goals to be satisfied (in some sequence) after the parent goal has been satisfied. Depending upon the sequence in which the conjoined subgoals are satisfied, different OR trees will be generated, each related to the AND/OR tree.

The nodes of the tree are labelled by all the outstanding subgoals. The arcs are labelled by the number of the resolving clause and the most general unifier. Prolog searches this tree in a depth-first manner, choosing the left-most subgoal to resolve away at each stage. The different environments of recursive calls are represented by different variables and different bindings for them.

The OR tree for the location example is given in figure 4-2.

Presenting the OR tree in different modes causes similar difficulties to presenting the AND/OR tree. The effect of the cut would be displayed by scribbling out the outstanding choice points and the trees they dominate, when the ! subgoal is satisfied.

They both have similar features and problems, with the following exceptions:

* The outstanding subgoals in the OR tree are carried along at each stage, unaltered. They tend to clutter up the tree.

* With the OR tree representation it is easier to see what subgoals are

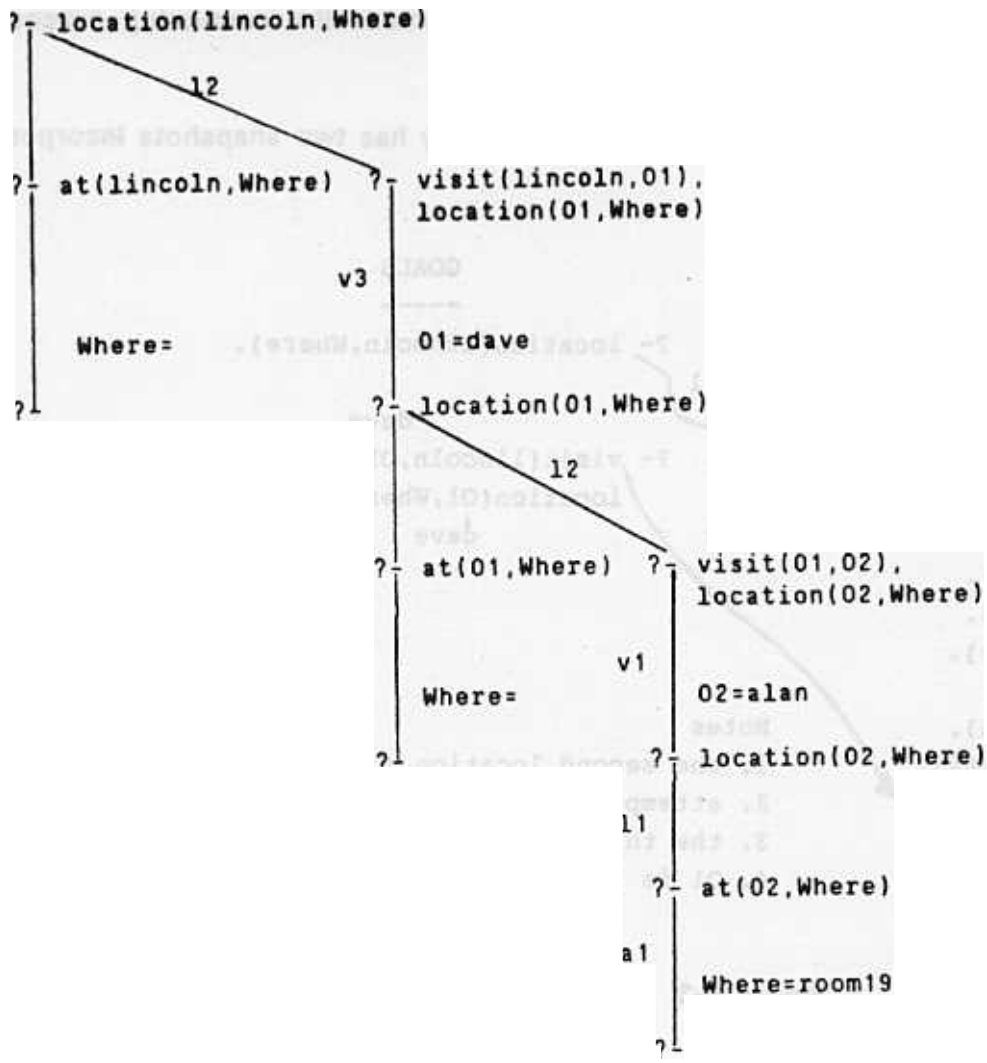

Figure 4-2: An OR Tree

outstanding and it is clearer when a call has been successful.

The main differences between the two sorts of trees is that the OR tree displays success and the outstanding subgoals in a more straightforward way at the cost of recording information (the outstanding subgoals) redundantly. However, as the OR tree is a collapsed AND/OR tree, the OR tree can be derived from the AND/OR tree.

4.3. The Arrow Story

The arrow story is introduced in 'Programming in Prolog' [Clocksin and Mellish 81]. It solves part of the problem of AND/OR trees and OR trees by providing a direct tie to the database.

Part of the OR tree is presented together with the database. In addition direct links (the arrow) are drawn between each subgoal and the resolving clause. This story tells, mainly in prose, about satisfaction of subgoals, left/right order of subgoals, top-bottom order of clauses, and backtracking. Subsequent snapshots document the continuing process of satisfying this goal.

An example is given in figure 4-3. This example actually has two snapshots incorporated in one diagram.

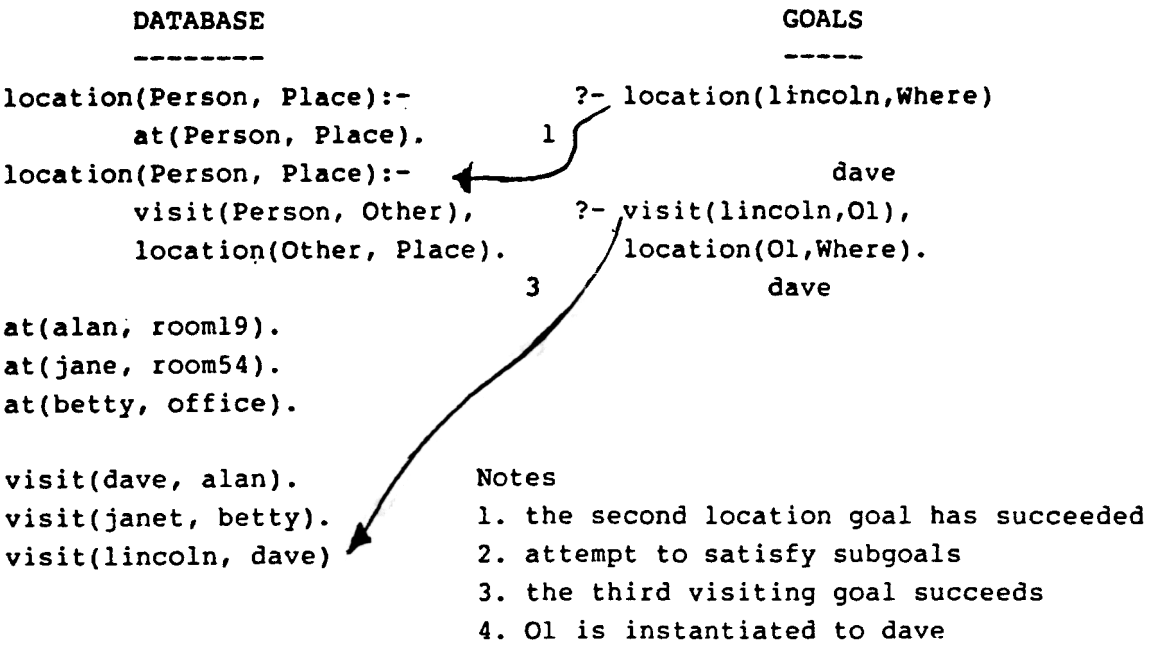

Figure 4-3: An Arrow Diagram

Probably the easiest way to present this story is with overhead transparencies, overlaying the arrows, goals and explanation for each snapshot or on the blackboard. On the standard v.d.u. the story could be displayed dynamically, though it might be difficult to represent the arrows. In text, each diagram is simple enough but a number of diagrams are needed and the full explanation might be tedious.

The detail of this presentation is good for a beginning student, especially the explicit linking of the selected subgoal with the resolving clauses and the display of the alternative clauses in the format of the program.

The search space is described as a sequence of snapshots: its tree structure is not exhibited. The passage of the arrow heads down the list of clauses provides an alternative explanation of backtracking and cut. All the outstanding subgoals are shown in each snapshot, as in the OR tree. It provides a window onto the OR tree.

Since only a single copy of each clause is displayed, it is not clear how different instantiations of the same clause would be represented. One possibility would be labelling the arrow with the unifier.

The arrow story also presents practical difficulties in the number of fingers and arrows needed in explanation of the links.

4.4. The Byrd Box Model

This is the story developed by Lawrence Byrd for the debugging aid he built for Prolog [Byrd 80, Bowen 82].

The Byrd box model comes in two parts: the boxes drawn in explanations and the output given during a Prolog session. The first part consists of drawings like that in figure 4-4. The second part consists of a dialogue like that in figure 4-5.

All the clauses for a particular predicate are enclosed in a 'box'. Arrows into and out of the box show how clauses can be entered (by CALL and REDO) and left (by EXITing on success, or by FAILure). This story can be used to demonstrate how the database of Prolog clauses is 'gone through' in finding the solution to a goal. It may be seen as representing the paths through a node of the AND/OR tree.

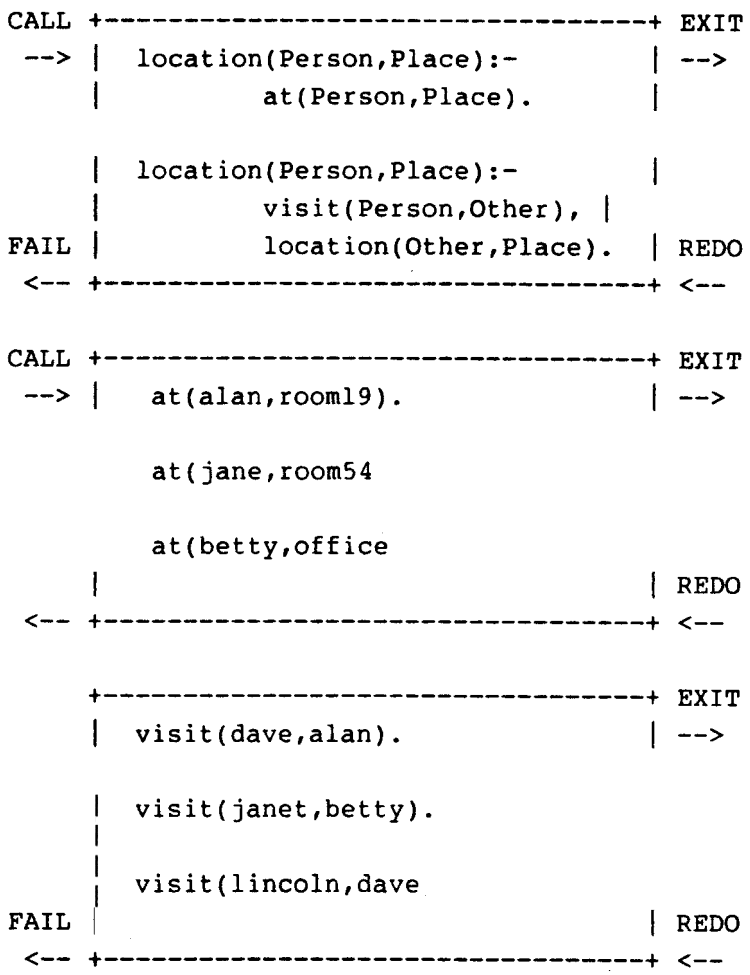

Figure 4-4: Three Byrd Boxes

The second part of the box model, the trace, essentially steps arounds the AND/OR tree. This trace or record of goals is needed to connect the boxes to the tree. It steps down the tree with CALL and REDO messages, and up the tree with EXIT and FAIL messages. An

incomplete version of the tree information, namely the depth of the nodes, is given by the unbracketted numbers in the trace. The boxes complement the trace obtained by focussing on a predicate. They help the user understand what causes the subgoals in the trace to change.

```
?- trace,location(lincoln,Where).
Debug mode switched on.
(1) 0 Call : location(lincoln,_38) ?
(2) 1 Call : at(lincoln,_38) ?
(2) 1 Fail : at(lincoln,_38)
(3) 1 Call : visit(lincoln,_108) ?
(3) 1 Exit : visit(lincoln,dave)
(4) 1 Call : location(dave,_38) ?
(5) 2 Call : at(dave,_38) ?
(5) 2 Fail : at(dave,_38)
(6) 2 Call : visit(dave,_126) ?
(6) 2 Exit : visit(dave,alan)
(7) 2 Call : location(alan,_38) ?
(8) 3 Call : at(alan,_38) ?
(8) 3 Exit : at(alan,room19)
(7) 2 Exit : location(alan,room19)
(4) 1 Exit : location(dave,room19)
(1) 0 Exit : location(lincoln,room19)

Where = room19

yes
```

Figure 4-5: A Byrd Box Trace

Individual Byrd boxes can be displayed fairly easily in all modes. Following the paths through the boxes is more difficult. Telling the story using the boxes alone gives a linear and local story: the trace must also be displayed. For this simple program the trace is very short. For others it is much longer. Options are usually offered to permit the user to focus on specified predicates, and to avoid tracing lower level goals. A lot of information is given in the trace although it is not easy to reconstruct the tree from this.

The Byrd box story has most of the disadvantages of the AND/OR tree representation, except that the EXIT messages tell you about the state of the goals after satisfaction and hence tell you about the output substitution. In addition, two pieces of information present in our drawing of the AND/OR tree are absent from the Byrd trace messages, namely the clauses being resolved with and the overall shape of the tree. The clauses are given in the boxes - the first part of the Byrd box model - but this information is not available on-line, and the clauses linked are not linked to the subgoals they satisfy. It is also difficult to see what subgoals are outstanding at any moment. The variables are renamed to numbers in order to separate the different environments of recursive calls, but the origins of the variables are not clear. No information is given about which parts of the tree are deleted by cuts.

5. Conclusion

It can be seen from the above that some stories contain more information than others. The extra information is paid for by the increase in complexity. For example, in the OR tree it is easy to see what subgoals remain at any point as the nodes are labelled with these. However, this is done at the cost of carrying unsatisfied subgoals forward in the current goal clause. The AND/OR tree is less redundant: each subgoal appears only once. However, working out which are the unsatisfied subgoals, and when the initial goal has succeeded, is more difficult.

None of the stories provide a clean and easy way of dealing with the cut. This is not, however, unexpected: if a nice clean story could be told about the cut then there would not be so many people struggling to replace it.

In considering the AND/OR tree and OR tree representations of the search space, it can be seen that they are closely related. The OR tree, however, incorporates some goal ordering constraints that the AND/OR tree does not have. It is suggested, therefore, that the AND/OR tree is the more fundamental representation of the two. All of the stories described above can be developed consistently from the AND/OR tree story of the search space. The complete model of Prolog to be represented in any story is:

* the AND/OR tree story of the search space;

* the database of clauses;

* the information for controlling search.

Presenting all this information in one single story leads to a confusing complexity for all but the simplest examples. An alternative solution is to make all the information relating to the model available, but to choose to show only a portion of it at any one time. This would necessitate deciding exactly what portion to present in blackboard or text mode, and providing other information as required. In computer mode, for example in a debugger, we could offer various options: the user could ask to see all matching clauses for a goal if this information was not given automatically. We present some parts of the story automatically and give access to the remainder. The actual story that we present will be determined by the aims and background of the users. As long as presentation in all modes is based on the same model, there should be consistency in the overlapping parts of different stories.

As bit-map displays come into more common use the difficulties of presentation in different modes will be reduced, though there will still be problems with reproducing, in the classroom, what is shown at the terminal. Video presentation might be one means of reducing this problem.

The main conclusions to be drawn here are:

* It is important to consider the model of Prolog that is to be used in teaching, and to select the most important features of this model for presentation.

  The story that is told about this model must be clear and consistent: and should provide a framework for understanding the model.

* Decisions must be made concerning the mode in which the story is to be told and the language that is used to tell it.

References

[Bowen 82] Bowen, D.L., Byrd, L., Pereira, F.C.N., Pereira, L.M. and Warren, D.H.D.
Decsystem-10 Prolog User's Manual.
Occasional Paper 27, Dept. of Artificial Intelligence, Edinburgh, November, 1982.

[Byrd 80] Byrd, L.
Understanding the control flow of PROLOG programs.
In Tarnlund, S. (editor), *Proceedings of the Logic Programming Workshop*, pages 127-38. , 1980.

[Clocksin and Mellish 81]
Clocksin, W.F. and Mellish, C.S.
Programming in Prolog.
Springer Verlag, 1981.

[du Boulay and O'Shea 78]
du Boulay, B and O'Shea, T.
Seeing the works: A strategy for teaching interactive programming.
In *Proceedings of the Workshop on Computing Skills and Adaptive Systems*. , Liverpool, March, 1978.
also available as DAI working paper no. 28.

[Kowalski 79] Kowalski, R.
Artificial Intelligence Series. Logic for Problem Solving.
North Holland, 1979.